Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage

James A. Jones and Mary Jean Harrold
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
{jjones,harrold}@cc.gatech.edu

Abstract

Software testing is particularly expensive for developers of high-assurance software, such as software that is produced for commercial airborne systems. One reason for this expense is the Federal Aviation Administration's requirement that test suites be modified condition/decision coverage (MC/DC) adequate. Despite its cost, there is evidence that MC/DC is an effective verification technique, and can help to uncover safety faults. As the software is modified and new test cases are added to the test suite, the test suite grows, and the cost of regression testing increases. To address the test-suite size problem, researchers have investigated the use of test-suite reduction algorithms, which identify a reduced test suite that provides the same coverage of the software, according to some criterion, as the original test suite, and test-suite prioritization algorithms, which identify an ordering of the test cases in the test suite according to some criteria or goals. Existing test-suite reduction and prioritization techniques, however, may not be effective in reducing or prioritizing MC/DC-adequate test suites because they do not consider the complexity of the criterion. This paper presents new algorithms for test-suite reduction and prioritization that can be tailored effectively for use with MC/DC. The paper also presents the results of a case study of the test-suite reduction algorithm.

1 Introduction

To facilitate the testing of evolving software, a test suite is typically developed for the initial version of the software, and reused to test each subsequent version of the software. As new test cases are added to the test suite to test new or changed requirements, or to maintain test-suite adequacy, the size of the test suite grows, and the cost of running it on the modified software (i.e., *regression testing*) increases.

Regression testing is particularly expensive for developers of high-assurance software, such as software that is produced for commercial airborne systems. One reason for this expense is the extensive verification of the software required for Federal Aviation Administration approval. Such approval includes the requirement that a test suite be adequate with respect to modified condition/decision coverage (MC/DC). For example, one of our industrial partners reports that for one of its products of about 20,000 lines of code, the MC/DC-adequate test suite requires seven weeks to run. Despite its cost, there is evidence that MC/DC is an effective verification technique. For example, a recent empirical study performed during the real testing of the attitude-control software for the HETE-2 (High Energy Transient Explorer) found that the test cases generated to satisfy the MC/DC coverage requirement detected important errors not detectable by functional testing.[1]

Researchers have investigated two approaches that maintain the same coverage as the original test suite for addressing the test-suite size problem:[2] test-suite reduction and test-suite prioritization. *Test-suite reduction* (also known as minimization) algorithms (e.g., [1, 7, 8, 10, 12, 13]) identify a reduced test suite that provides the same coverage of the software as the original test suite. *Test-suite prioritization* algorithms (e.g., [4, 5, 11]) identify an ordering of the test suite according to some criteria.

Existing test-suite reduction and prioritization techniques consider a set of test-case coverage criteria (e.g., statements, decisions, definition-use associations, or specification items), other criteria such as risk or fault-detection effectiveness, or combinations of these criteria. As we discuss in Section 4, for test-suite reduction and prioritization based on coverage there are important differences between coverage criteria, such as statement, and MC/DC. Thus, ex-

[1] Private communication with Nancy Leveson of MIT.

[2] Another related approach that addresses the test-suite size problem is test selection (e.g., [2, 9]), which selects a subset of the test suite that will execute code or entity changes; this test suite, however, may not provide the same coverage as the original test suite.

isting techniques may not be effective for use in reducing or prioritizing MC/DC-adequate test suites because they do not consider the complexities of the criterion. However, because of the enormous testing expense that users of MC/DC can incur, effective test-suite reduction and prioritization techniques that can be applied to MC/DC-adequate test suites could provide significant savings to developers who use this powerful test-coverage criterion.

This paper presents new algorithms for test-suite reduction and prioritization that can incorporate aspects of MC/DC effectively. Unlike existing algorithms, when making decisions about reducing or ordering a test suite, our algorithms consider the complexities of MC/DC. This paper also presents a case study that evaluates the effectiveness of the test-suite reduction algorithm on a subject program and a number of test suites.

2 Modified Condition/Decision Coverage

MC/DC is a stricter form of decision (or branch) coverage. For decision coverage, each decision statement must evaluate to `true` on some execution of the program and must evaluate to `false` on some execution of the program.[3] MC/DC, however, requires execution coverage at the condition level. A *condition* is a Boolean-valued expression that cannot be factored into simpler Boolean expressions. For example, the partial program P in Figure 1 (a) contains a decision statement, S1, that has three conditions: A, B, and C.

MC/DC requires that each condition in a decision be shown by execution to independently affect the outcome of the decision [3]. Each possible evaluation of that decision produces a *truth vector*—a vector of the Boolean values resulting from the evaluation of the conditions in that decision. For example, "TTF" in Figures 1 and 2 for statement S1 is a truth vector in which condition A evaluates to `true`, B evaluates to `true`, and C evaluated to `false`.

An *MC/DC pair* is a pair of truth vectors, each of which causes a different result for the decision statement, but that differ only by the value of one condition. For example, the truth vectors "FTT" and "FTF" for condition C in Figure 2 differ only in the evaluation of C and cause different evaluations of the decision. Thus, these two truth vectors comprise an MC/DC pair for condition C. The truth vector "FTF" is called "FTT"'s *mate truth vector*, and vice versa.

A *truth value* for a condition c is the set of truth vectors belonging to the MC/DC pairs for c that cause c's decision to evaluate to a particular result. Figure 2 illustrates C's truth values: {FTT, TFT, TTT} for C(T) and {FTF, TFF, TTF} for C(F). Note that, although the truth vector "FFF" causes C's decision to evaluate to `false`, it is not an element of the C(F) truth value because it is not part of an MC/DC pair for C.

In some cases, there can be more than one MC/DC pair for a condition. Figure 1 (b) shows, for example, that there is only one MC/DC pair for conditions A, B, D, and E but

[3] If no inputs cause these `true` or `false` evaluations for the predicate, the decision is *infeasible* and cannot be covered by any test case.

Program P

S1: if (A or B) and C

⋮

S2: if (not (D and E)) and F

⋮

end P

(a) Partial program P

S1: if (A or B) and C

A(T)	A(F)
TFT	FFT

B(T)	B(F)
FTT	FFT

C(T)	C(F)
FTT	FTF
TFT	TFF
TTT	TTF

S2: if (not(D and E)) and F

D(T)	D(F)
FTT	TTT

E(T)	E(F)
TFT	TTT

F(T)	F(F)
FTT	FTF
TFT	TFF
FFT	FFF

(b) Truth vectors and MC/DC pairs for S1, S2

	S1	S2
T1	TFT	TFT
T2	FTF	FTF
T3	FFT	FTT
T4	FTT	TFF
T5	TFF	TTT
T6	FTT	TTT

(c) Test cases and their truth vector coverage

S1: if (A or B) and C

A(T)	A(F)
T1	T3

B(T)	B(F)
T4, T6	T3

C(T)	C(F)
T4, T6	T2
T1	T5
--	--

S2: if (not(D and E)) and F

D(T)	D(F)
T3	T5, T6

E(T)	E(F)
T1	T5, T6

F(T)	F(F)
T1	T2
T3	T4
--	--

(d) MC/DC pairs by test case coverage in S1, S2

Figure 1. (a) Partial program P; (b) its truth vectors, organized by condition, truth value, and MC/DC pair; (c) test cases and their coverage; (d) subfigure b with the truth vectors replaced by the test cases that cover them.

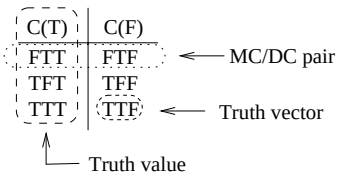

Figure 2. Condition C (Figure 1) and its constituent parts.

there are three MC/DC pairs for conditions C and F, each of which can be used to show the independence of the condition. For condition C, for example, there are three MC/DC pairs; we denote them as C_i, where i is the row number in the table for C. Because MC/DC requires that MC/DC pairs be covered, the coverage of truth vectors "FTT" and "TFF", although satisfying C_1(T) and C_2(F) respectively for condition C, does not satisfy the MC/DC criterion for C.

3 Test-suite Reduction and Prioritization

The *test-suite reduction problem* can be stated as:[4]

> *Given*: Test suite, T, a set of test-case requirements, $r_1, r_2, \ldots, r_n$, that must be satisfied to provide the desired test coverage of the program.
>
> *Problem*: Find $T' \subset T$ such that $(\forall T'')$ $(T'' \subset T)$ (T'' satisfies all r_is) $[|(T'| \leq |T''|]$.

Given subsets of T, $T_1, T_2, \ldots, T_n$, one associated with each of the r_is such that any one of the test cases t_j belonging to T_i can be used to test r_i, a test suite that satisfies all r_is must contain at least one test case from each T_i. Such a set is called a hitting set of the T_i. Maximum reduction is achieved with the minimum cardinality hitting set of the T_is. Because the problem of finding the minimum cardinality hitting set is intractable [6], test-suite reduction techniques must be heuristics.

The *test-suite prioritization problem* can be stated as:[5]

> *Given*: Test suite, T, a set of permutations of T, PT, a function, f, from PT to the real numbers.
>
> *Problem*: Find $T' \in PT$ such that $(\forall T'')$ $(T'' \in PT)$ $[f(T') \geq f(T'')]$.

PT represents the set of possible orderings (prioritizations) of T, and f is a function that, when applied to an ordering, yields an evaluation of that ordering. Depending on the choice of f, the test-case prioritization problem may be intractable or undecidable. For example, given a function f that quantifies whether a test suite achieves statement coverage at the fastest rate possible, an efficient solution to the prioritization problem would provide an efficient solution to the knapsack problem [6]. Similarly, given a function f that quantifies whether a test suite detects faults at the fastest rate possible, a precise solution to the prioritization problem would provide a solution to the halting problem. In such cases, prioritization techniques must be heuristics.

Test-suite reduction and prioritization algorithms share a number of common attributes. First, these algorithms input a test suite, T, and output a new test suite, T'. In constructing T', the algorithms evaluate each test case for its *contribution* or *goodness* based on some characteristics of the program under test, $\mathcal{P}$, some characteristics of the test cases in T, and the goal of T'. Although a variety of characteristics have been considered, we can classify them as either types of coverage or types of cost. *Types of coverage* include $\mathcal{P}$'s requirements, code-based criteria such as statement, branch, and data-flow, and risks, age, or error-proneness of components of $\mathcal{P}$. *Types of cost* include execution time for $\mathcal{P}$ with a test case, set-up time for preparing to run a test case, and any financial cost associated with running a test case (especially in the case of simulations). The algorithms evaluate test cases in T based on some combination of these characteristics. For example, References [7, 10] use various types of coverage for test-suite reduction. For each program entity, E, this technique uses the number of test cases that cover E when making decisions about which test cases to include in T'. For another example, References [4, 11] use various types of coverage, along with error-proneness of modules, to select an ordering of test cases for T'.

Second, test-suite reduction and prioritization algorithms may vary in the frequency with which they recompute the contribution of a test case. One approach computes this contribution once, and uses it to select test cases for inclusion in or exclusion from T'. Rothermel et al. [11] refer to this frequency as *total*, and use it to compute total-based prioritization. Another approach recomputes the contribution after test cases are included in or excluded from T' based on the additional contribution of the test case to T'. Rothermel et al. [11] refer to this frequency as *additional*, and use it to compute additional-based prioritization. In discussing our algorithms, we use Rothermel et al.'s terminology to refer to techniques for computing a test case's contribution: total refers to an a priori evaluation in which test cases are evaluated based only on the characteristics of the test case, whereas additional refers to an evaluation that considers the current state of the reduction or prioritization as well as characteristics of the test case. In the case of additional evaluation, the frequency with which test cases are reevaluated may vary, and reevaluation may occur after n iterations.

Third, an algorithm can use a test case's contribution for determining the next test case to add to T' (for both reduction and prioritization) or, alternatively, for determining the next test case to remove from T (for reduction). We refer

[4] Variation of the problem presented in Reference [7, p. 272].

[5] Problem presented in Reference [11, p. 3].

to the former as a *build-up* technique and to the latter as a *break-down* technique. In a build-up technique, the algorithm begins with an empty T' and adds test cases to it, whereas in a break-down technique, the algorithm begins with T and removes test cases from it to get T'. One important advantage of a break-down approach for test-suite reduction is that the algorithm can be stopped at any time during the reduction process, and the remaining test suite (i.e., $T - \{\text{removed test cases}\}$) provides coverage of the test-case requirements. This coverage can be important if manual intervention is required in making decisions for test-case removal. In contrast, a build-up approach can guarantee coverage only when the algorithm terminates.

Combinations of these attributes—computation of test-case contribution, frequency of (re)evaluation of test-case contribution, and method of constructing T'—can produce a number of different algorithms. In the next section, we describe two algorithms that we have developed that consider the complexities of MC/DC.

4 Test-Suite Reduction and Prioritization for MC/DC

In this section, we first discuss the problems with using existing test-suite reduction and prioritization algorithms for MC/DC. Then, we present two new algorithms—one for test-suite reduction and one for test-suite prioritization—that consider the complexities of MC/DC.

4.1 Using Existing Algorithms for MC/DC

For test-suite reduction and prioritization, there are two important differences between coverage criteria, such as statement, and MC/DC. First, coverage criteria, such as statement, require that every entity (e.g., statement) be covered by the test suite. For these criteria, a test case either covers an entity or does not cover the entity; we call such criteria *single-entity* criteria. In contrast, MC/DC requires that every condition be covered by the execution of an MC/DC pair. For MC/DC, a test case may cover an MC/DC pair but, more often, a test case contributes to the coverage of a condition by covering *only one* of its truth vectors; we call such criteria *multiple-entity* criteria. Second, coverage criteria, such as statement, have only one way to be covered—the entity (e.g., statement) must be executed by one of the test cases in the test suite. For MC/DC, there may be several MC/DC pairs associated with a condition, and executing any of them provides coverage of the condition. For example, in partial program P of Figure 1(a), condition F has three possible MC/DC pairs, and coverage of only one of them is necessary to satisfy MC/DC for F.

Because of these differences, existing test-suite reduction and prioritization techniques, which were developed for single-entity criteria, are not sufficient when applied to multiple-entity criteria such as MC/DC. Existing test-suite reduction and prioritization techniques require an association between entities to be covered and test cases in the test suite. In attempting to use the existing techniques for MC/DC, the major problem is the identification of the entities to be covered. To illustrate the problem, consider two possible approaches for test-suite reduction; approaches for test-suite prioritization have similar problems but, given space constraints, will not be discussed.

Condition	Test-Case Pair
A	{(T1, T3)}
B	{(T3, T4), (T3, T6)}
C	{(T2, T4), (T2, T6), (T1, T5), (T1, T7)}
D	{(T3, T5), (T3, T6), (T3, T7) }
E	{(T1, T5), (T1, T6), (T1, T7) }
F	{(T1, T4), (T2, T3) }

Table 1. MC/DC pairs and associated test-case pairs for program P of Figure 1.

Under the first approach, for test-suite reduction, the r_is are the MC/DC pairs. The r_is would be, for example (Figure 1), MC/DC pair ("TFT", "FFT") for A, and ("FTT", FTF"), ("TFT", "TFF"), and ("TTT", "TTF") for C. Because, in most cases, two test cases are required to cover an MC/DC pair, we may not be able to get a relationship between test cases and MC/DC pairs. In fact, for the MC/DC pairs in program P, no MC/DC pair is covered by a single test case. Thus, we cannot associate individual test cases with MC/DC pairs. Because we cannot get this association of MC/DC pairs and test cases, we cannot use MC/DC pairs as the coverable entities for the reduction.

Under the second approach, for test-suite reduction, the r_is are the truth vectors. The r_is would be, for example, truth vectors "TFT" and "FFT" for A, and truth vectors "FTT", FTF", "TFT", "TFF", "TTT", and "TTF" for C. With this approach, we can associate individual test cases with truth vectors, and MC/DC is like the class of single-entity criteria in that a test case covers (or does not cover) each entity. This approach, however, is not sufficient for test-suite reduction because MC/DC requires that an MC/DC pair of truth vectors be covered. For example, $\mathrm{C(T)}_1$ may be satisfied by executing truth vector "FTT" and $\mathrm{C(F)}_2$ may be satisfied by executing truth vector "TFF." Both true and false entities are covered. However, MC/DC is not satisfied for this condition because the truth vectors do not constitute an MC/DC pair.

A naive approach for providing a test-suite reduction algorithm for MC/DC is to associate, with each MC/DC pair, the *pairs* of test cases that cover the pair. Given this association, we can apply Harrold, Gupta, and Soffa's (HGS) algorithm for reduction of the test suite [7].

Table 1 shows the association between MC/DC pairs and test-case pairs for our example program P (Figure 1a). The HGS algorithm would be applied to theses test pairs instead of the individual test cases. This application of the HGS algorithm would not take advantage of the intersection of test-case pairs in the reduction process. The resulting T is $\{$T1, T3, T4, T5, T6$\}$.

Although this approach can provide some reduction in the test suite, the algorithm does not consider individual test cases and the contribution they make to MC/DC. We believe, and our case study suggests, that we can achieve more reduction in test-suite size using an algorithm that considers the partial coverage of the MC/DC pairs by individual test cases. The next section presents this new algorithm and the following section presents a test-suite prioritization algorithm that considers individual test cases and the contribution they make to MC/DC.

4.2 New Test-suite Reduction Algorithm

Our test-suite reduction algorithm bases its contribution computation on MC/DC pairs, utilizes an additional approach that recomputes the contribution of test cases after each test case is selected, and uses a break-down approach that removes the weakest test case from the test suite.

To describe test-suite reduction, we define two important terms: test-case redundancy and test-case essentiality. A test case, t, is *redundant* with respect to a set of test cases, T, if the set of test-case requirements covered by t is a subset of the set of test-case requirements covered by T$-\{$t$\}$. To eliminate some test case, t, from a test suite T and retain the coverage of T, t must be redundant with the remaining test cases in T. A test case, t, is *essential* with respect to a set of test cases, T, if the set of test-case requirements covered by t is not a subset of the set of test-case requirements covered by T$-\{$t$\}$. A test case that uniquely covers any test-case requirement must be present in the reduced test suite to retain coverage of that requirement. Every test case, t, in a test suite, T, can be identified as either essential or redundant with respect to the remaining test cases in T.

More precisely, for MC/DC, a test case that uniquely covers a truth value of any covered condition (i.e., a condition with a covered MC/DC pair) after removing any uncovered MC/DC pairs is essential. Figure 3 demonstrates this process: first, we view the test-case coverage of the MC/DC pairs, as shown in (a); then, we remove any MC/DC pairs that are uncovered, as these do not contribute to the coverage of the condition, as shown in (b) with the removal of the second MC/DC pair; and finally, we identify the test cases that uniquely cover a test value, as shown in (c) with the dashed box around the pair of occurrences of T1. This process is applied to all covered conditions. Conditions that are uncovered by the original test suite are handled differently; we explain this in Section 4 (Other considerations).

X(T)	X(F)
T1	T2
T3	--
T1	T4

(a)

X(T)	X(F)
T1	T2
T1	T4

(b)

X(T)	X(F)
T1	T2
T1	T4

(c)

Figure 3. (a) Example condition shown with test-case coverage, (b) removal of uncovered MC/DC pairs, (c) identification of essential test cases.

Our algorithm first identifies essential test cases; the remaining test cases in the test suite are redundant with some other set of test cases in the suite. The algorithm then iteratively identifies the *weakest* test case (the test case contributing the least to coverage of the test-case requirements) of the test case that have not been marked as essential, eliminates that test case, and identifies new essential test cases. When all test-case requirements covered by the original test suite are covered by the set of essential test cases, the algorithm halts, and the essential test cases are returned as the reduced test suite.

Our algorithm `ReduceSuite`, shown in Figure 4, can be applied to both single-entity and multiple-entity criteria. `ReduceSuite` inputs (1) $allTests$—the test cases in the test suite, annotated with the conditions that they cover, and (2) $allConditions$—the conditions in the program under test, $\mathcal{P}$, annotated with the test cases that cover them. The algorithm outputs the test cases comprising the reduced test suite. Step 1 is executed once at the beginning of the algorithm, and then Steps 2, 3, and 4 are executed iteratively until $allConditions$ is empty.

Step 1: Eliminate uncovered MC/DC pairs. In Step 1 (lines 2-4), any MC/DC pairs that are not covered by $allTests$ are removed from $allConditions$. These MC/DC pairs cannot contribute to the MC/DC coverage. For example, the MC/DC pairs (C_3(T),C_3(F)) and (F_3(T),F_3(F)) [6] in conditions C and F, respectively, in Figure 1 are removed. Figure 5, which is modified from Figure 1(d), illustrates the state of the example after first iteration of Step 1 of `ReduceSuite`.

Step 2: Identify essential test cases. In the second step (lines 6-19), `ReduceSuite` identifies the test cases that are essential with respect to $allTests$. The algorithm examines each condition, c, in $allConditions$ to identify any test case that uniquely covers either c's `true` truth value or c's `false` truth value (lines 6-9). When the algorithm finds such a test case, t, it adds t to $essentialTests$ and removes t from $allTests$ (lines 10-11). Note that the $essentialTests$ set is kept as a convenience rather than a

[6] Recall from Section 2, that C_i is the ith MC/DC pair in condition C.

```
algorithm   ReduceSuite(allTests, allConditions)
input       allTests: set of test cases for P
            allConditions: set of conditions for P
output      essentialTests: reduced test suite
 begin ReduceSuite
1.   essentialTests = φ
2.   for each condition, c, in allConditions do                /*Step 1*/
3.     remove all uncovered MC/DC pairs from c
4.   endfor
5.   do
6.     for each condition, c, in allConditions do              /*Step 2*/
7.       for x in {true,false} do
8.         if the (# of test cases covering all of the x truth vectors in c)
               = 1 then
9.           t = the test covering x truth vectors in c
10.          essentialTests = essentialTests ∪ {t}
11.          allTests = allTests − {t}
12.          for each truth vector, v, covered by t do
13.            if (the set of test cases covering v's mate truth vector in
                   its MC/DC pair) ∩ essentialTests ≠ φ then
14.              allConditions = allConditions − {condition
                     to which v belongs}
15.            endif
16.          endfor
17.        endif
18.      endfor
19.    endfor
20.    for each test, t, in allTests                           /*Step 3*/
21.      t.contribution = 0
22.    endfor
23.    for each condition, c, in allConditions do
24.      for x in {true,false} do
25.        indivContrib = indivContrib + 1.0/(# of test cases
               covering all of the x truth vectors in c)
26.        for each x truth vector, v, in c do
27.          for each test, t, covering v do
28.            t.contribution = t.contribution + indivContrib
29.          endfor
30.        endfor
31.      endfor
32.    endfor
33.    t = test in allTests with lowest contribution           /*Step 4*/
34.    allTests = allTests − {t}
35.    for each truth vector, v, that t covers do
36.      v.coveringTests = v.coveringTests − {t}
37.      if (the MC/DC pair, p, to which v belongs becomes
             uncovered) then
38.        remove p from its condition
39.      endif
40.    endfor
41.  while (allConditions ≠ φ)
42.  return essentialTests
 end ReduceSuite
```

Figure 4. `ReduceSuite`: MC/DC adequate test suite reduction.

necessity. The essential test cases could also be marked essential in the $allTests$ set, but for ease and speed, a separate set is used—this is not a build-up algorithm. The algorithm then examines each truth vector, v, that t covers to see whether any of the test cases that cover v's mate truth vector are essential (line 13); this indicates a covered MC/DC pair, and thus a covered condition. When the algorithm finds

A(T)	A(F)
T1	T3

B(T)	B(F)
T4, T6	T3

C(T)	C(F)
T4, T6	T2
T1	T5

D(T)	D(F)
T3	T5, T6

E(T)	E(F)
T1	T5, T6

F(T)	F(F)
T1	T2
T3	T4

allTests = {T1,T2,T3,T4,T5,T6}
allConditions = {A,B,C,D,E,F}
essentialTests = {}

Figure 5. Example after first iteration of Step 1 of `ReduceSuite`.

such a case, it removes the condition to which v belongs from $allConditions$ (line 14).

In the example of Figure 1, because T1 is the only test case that covers A(T), `ReduceSuite` identifies it as an essential test case, removes it from $allTests$, and adds it to $essentialTests$; Likewise, the algorithm identifies T3 as essential, removes it from $allTests$ and adds it to $essentialTests$. Next, the algorithm identifies condition A as covered by $essentialTests$ and removes it from $allConditions$. Figure 6 shows the state of the example after the first iteration of Step 2. The dotted boxes denote the essential test cases and the slashed condition denotes a condition covered by $essentialTests$.

A(T)	~~A(F)~~
T1	T3

B(T)	B(F)
T4, T6	T3

C(T)	C(F)
T4, T6	T2
T1	T5

D(T)	D(F)
T3	T5, T6

E(T)	E(F)
T1	T5, T6

F(T)	F(F)
T1	T2
T3	T4

allTests = {T2,T4,T5,T6}
allConditions = {B,C,D,E,F}
essentialTests = {T1,T3}

Figure 6. Example after first iteration of Step 2 of `ReduceSuite`.

Step 3: Assign test-case contributions. After the algorithm identifies all essential test cases and removes them from $allTests$ (Step 2), each remaining test case, t, is redundant with $allTests - \{t\}$. Thus, the algorithm can remove any test case from $allTests$ and retain coverage

of $\mathcal{P}$. In Step 3 (lines 20-32), the algorithm attempts to find the test case that contributes least to the coverage in $allTests$. The algorithm does this by first initializing the $contribution$ of each test case to zero (lines 20-22). Then, for each condition, c, the algorithm increments the contribution of each test case that covers each of c's `true` and `false` truth values, tv, by $(1.0/\text{the number test cases covering } tv)$ (lines 23-32).

In the example, the contributions given to the test cases in $allTests$ are shown in Table 2. To illustrate, the contribution of $1.3\overline{3}$ for T4 is achieved by summing the individual contribution scores for its coverage of B(T), C(T), and F(F) (i.e., 0.5, $0.\overline{3}$, and 0.5, respectively).

Test Case	Contribution Weight
T2	1.00
T4	$1.3\overline{3}$
T5	1.50
T6	$1.8\overline{3}$

Table 2. Test cases for example and associated contribution weights assigned during Step 3.

Step 4: Discard weakest test case. In Step 4 (33-40), the algorithm eliminates the weakest test case—the test case that contributes least to the coverage of $\mathcal{P}$, according to the heuristic given in Step 3. The test case with the lowest contribution in $allTests$, t, is removed from $allTests$ (line 34). For each of the truth vectors, v, that were covered by t, the algorithm removes t from the set of test cases covering v (lines 35-40). If the MC/DC pair to which v belongs becomes uncovered as a result of discarding t, the algorithm removes the MC/DC pair from its condition (37-39). Figure 7 shows the state of the example after the test case with the lowest contribution, T2, is discarded.

A(T)	~~A(F)~~
~~T1~~	T3

D(T)	D(F)
T3	T5, T6

B(T)	B(F)
T4, T6	T3

E(T)	E(F)
T1	T5, T6

C(T)	C(F)
T4, T6	--
T1	T5

F(T)	F(F)
T1	--
T3	T4

allTests = {T4,T5,T6}
allConditions = {B,C,D,E,F}
essentialTests = {T1,T3}

Figure 7. Example after first iteration of Step 4 of `ReduceSuite`.

Steps 2, 3, and 4 are repeated until the $allConditions$ set is empty, and thus all conditions in the $\mathcal{P}$ are covered by a set of essential test cases. In the next iteration of the algorithm, T4 and T5 are identified as essential. They are thus removed from $allTests$ and added to $essentialTests$. At this point, conditions B, C, D, E, and F are covered by the test cases in $essentialTests$ and are thus removed from $allConditions$. T6 is discarded as it is the only test in $allTests$, and consequently has the lowest contribution. The loop halts because $allConditions$ is empty and $essentialTests$ is returned as the reduced test suite. Figure 8 shows the state of the example after the next iteration of Steps 2, 3, and 4.

A(T)	~~A(F)~~
T1	T3

D(T)	~~D(F)~~
T3	T5

B(T)	~~B(F)~~
T4	T3

E(T)	~~E(F)~~
T1	T5

C(T)	~~C(F)~~
T4	--
~~T1~~	T5

F(T)	~~F(F)~~
T1	--
~~T3~~	T4

allTests = {}
allConditions = {}
essentialTests = {T1,T3,T4,T5}

Figure 8. Example after second iteration of Steps 2, 3, and 4 of `ReduceSuite`.

Other considerations. To simplify our discussion, we presented our algorithm, `ReduceSuite`, for test suites that are MC/DC-adequate. However, with minor modification, the algorithm can reduce test suites that are not MC/DC-adequate. In this case, there are uncovered or partially covered MC/DC pairs. For uncovered or partially covered conditions, in Steps 1, the algorithm eliminates all MC/DC pairs that have no coverage: neither truth vector in the MC/DC pair is covered by any test case. If all MC/DC pairs are removed from a condition, then the condition is not covered at all, and it is removed from $allConditions$. For partially covered conditions, we define essentiality slightly differently. A test case is considered to be essential if it uniquely covers a truth *vector* of a partially covered condition. Figure 9 shows test cases T1, T2, and T3—all of which are essential. We treat partially covered conditions in this way to ensure that if, at some later time, new test cases are added to the reduced test suite, that test suite will preserve the original test suite's potential for coverage. For example, if partially covered conditions were treated in the same manner as covered conditions, in Figure 9, we would need only one test case in $\{$T1, T3$\}$ to be in the reduced test suite. Suppose that T1, but not T3, was chosen to be in the

reduced test suite. If a test case was added to the test suite that covered X(F) in the third MC/DC pair, coverage would not be achieved for this condition.

X(T)	X(F)
T1	--
--	T2
T3	--

Figure 9. Example partially covered condition with essential test cases T1, T2, and T3.

The MC/DC criterion also specifies that all entry and exit points, and all case statements in the program be exercised. A simple extension to `ReduceSuite` handles entry and exit points and case statements by stating that essentiality is achieved when one test case covers each of these entities.

4.3 New Test-suite Prioritization Algorithm

Like our reduction algorithm, our test-suite prioritization algorithm bases its contribution computation on MC/DC pairs, and utilizes an additional approach that recomputes the contribution of test cases after each test case is selected. However, instead of the test-case evaluation being based on the uniqueness of program-entity coverage, this algorithm uses a simpler evaluation based on additional MC/DC pairs covered. In contrast to our test-suite reduction, the test-suite prioritization algorithm uses a build-up approach that adds the "strongest" test case to the new test suite.

Step 1: Initialization. In Step 1, our algorithm `PrioritizeSuite`, shown in Figure 10, first selects the test case in the test suite that has the highest entity coverage (sum of the test vectors, procedure entries, procedure exits, and cases covered) (line 3). This test case is placed into an ordered list, $orderedTestSuite$, as the first test case and removed from the set containing the original test suite, $allTests$. The algorithm augments each program entity with a flag that denotes its coverage by a test case in the unprioritized test suite (lines 1,4,11,18).

Step 2: Additional prioritization. Then, algorithm iterates over Step 2 until $allTests$ is empty. In each iteration, all test cases in $allTests$ are assigned a contribution value (lines 7-9). The contribution, or goodness, for test case t is evaluated based on the number of MC/DC pairs that are completed by t, plus the number of entries, exits, and case statements covered that are still uncovered by the test cases in $orderedTestSuite$. The next test case is chosen by selecting the test case with the highest contribution value. However, if all test cases' contribution values are zero, then

```
algorithm  PrioritizeSuite(allTests, allEntities)
input      allTests: set of test cases for P
           allEntities: set of entities for P
output     orderedTestSuite: prioritized test suite
 begin PrioritizeSuite
1.   mark all entities in allEntities uncovered          /* Step 1 */
2.   orderedTestSuite = empty array of size allTests
3.   orderedTestSuite[0] = {t ∈ allTests | ∀t' ∈ allTests, entity
           coverage of t' ≤ entity coverage of t}
4.   mark all entities in allEntities, covered, that are covered by t
5.   i = 1
6.   while allTests ≠ φ do                              /* Step 2 */
7.     for each test, t, in allTests do
8.       t.contribution = the sum of the # of MC/DC pairs
             completed and the number of entries, exits, and cases
             covered
9.     endfor
10.    if the highest contribution = 0 then
11.      mark all entities in allEntities uncovered
12.      nextTest = {t ∈ allTests | ∀t' ∈ allTests, entity
             coverage of t' ≤ entity coverage of t}
13.    else
14.      nextTest = {t ∈ allTests | ∀t' ∈ allTests,
             t'.contribution ≤ t.contribution}
15.    endif
16.    orderedTestSuite[i] = nextTest
17.    allTests = allTests − {nextTest}
18.    mark all entities in allEntities, covered, that are covered by
             nextTest
19.    i = i + 1
20.  endwhile
21.  return orderedTestSuite
 end PrioritizeSuite
```

Figure 10. `PrioritizeSuite`: MC/DC based test-suite prioritization.

the as of yet $orderedTestSuite$ has the same coverage as the original test suite. To continue the prioritization process, we reset all coverage flags for the program entities and select the next test case add to $orderedTestSuite$ based on the highest entity coverage (lines 10-12). The test case chosen, $nextTest$, is removed from $allTests$ and placed into the next available spot in $orderedTestSuite$, and all entities that are covered by $nextTest$ are marked `covered` (lines 16-18). This process iterates until all test cases in $allTests$ are consumed, and thus $orderedTestSuite$ contains all of the test cases in the original test suite. Finally, $orderedTestSuite$ is returned.

Note that a build-up, additional reduction algorithm could be implemented by augmenting this algorithm with the ability to detect when coverage of the $orderedTestSuite$ reaches the coverage of $allTests$. For example, this approach could be implemented in this algorithm by initializing a counter to the total number of conditions and entities that are covered by the initial test suite and decrementing this value by the number of additional conditions and entities covered by each selected test case. When this value reaches zero, the $orderedTestSuite$ can be used as a reduced test suite.

5 Case Study

This section describes the first of a number of planned studies to evaluate the effectiveness of our test-suite reduction and test-suite prioritization algorithms.

To investigate the effectiveness of our algorithms, we implemented prototypes of the reduction and prioritization algorithms. Both prototypes are written in C++. Much of the code is used in both prototypes because they both utilize the same data structures. The combined code for the prototypes consists of 4560 lines of code. We performed the studies on a Pentium III, 733 MHz computer.

Our subject for the study is the `Space` program,[7] which consists of 9564 lines of C code (6218 executable). `Space` is an interpreter for an array definition language (ADL). We also used 35 faulty versions of the program, each containing one fault, and the *base* version, which is assumed for purposes of studies to contain no faults. We instrumented `Space` by hand for MC/DC by placing a probe at every condition, procedure entry, procedure exit, and case statement in a **switch**. We also have a test pool of 13,585 test cases. From this test pool, 1000 randomly sized, randomly generated (plus additional test cases to reach near-decision coverage) test suites were extracted. This subject and these test suites suites have been used in similar studies (e.g., [5, 10, 11]); these references provide additional details about this subject. These test suites are near decision-coverage-adequate—only infeasible or extremely difficult to execute branches (such as those controlling an out-of-memory error condition) were not executed. These test suites covered 80.7% to 81.6% of the 539 conditions in the 489 decisions. The test suites ranged in size from 159 to 4712 test cases.

To evaluate our test-suite reduction algorithm, we applied our test-suite reduction prototype for MC/DC to each of the 1000 test suites. For each test case in the union of all test suites (13,585 test cases), `Space` was run to acquire a test-case requirement coverage report. We then executed our algorithm with the coverage information for all test cases in each of the test suites.

Figure 11 is a scatterplot where each point represents a test suite that has been reduced and is positioned along the horizontal axis and vertical axis based on its original test suite size and reduced test suite size, respectively. The average size of the reduced test suites is 111.8, the standard deviation is 3.44, the smallest test suite is 106 test cases, and the largest test suite is 125. Figure 11 shows that the sizes of the reduced test suites for the smaller original test suites is slightly larger than the sizes of the reduced test suites for the larger original test suites. We suspect that this trend is due to the paucity of test cases with potentially high contributions to test-case requirements coverage in the smaller test suites. We plan additional studies to investigate this suspicion. Figure 12 shows the time required to reduce the test

[7] Alberto Pasquini, Phyllis Frankl, and Filip Vokolos provided the Space program and many of its test cases. Chengyun Chu assisted with further preparation of the Space program and development of its test cases.

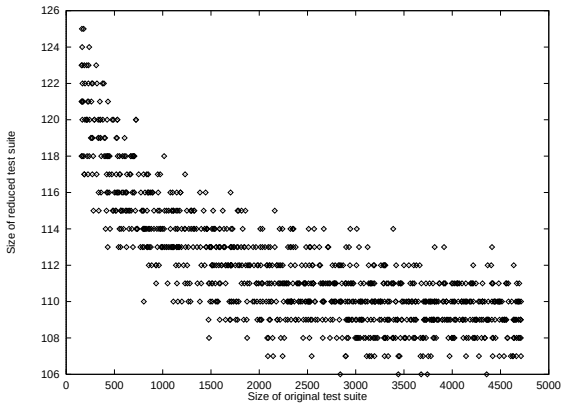

Figure 11. Sizes of the reduced test suite versus the sizes of the original test suite.

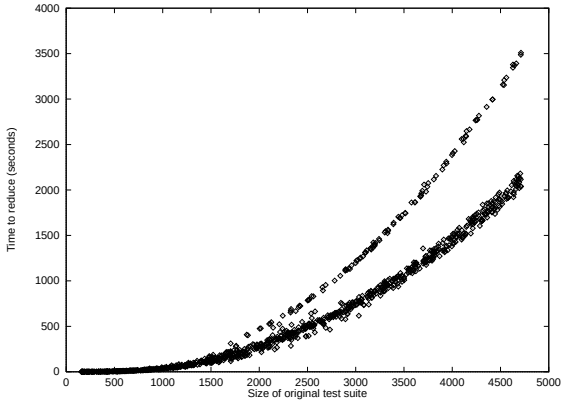

Figure 12. Time to run the reduction algorithm versus the sizes of the original test suite.

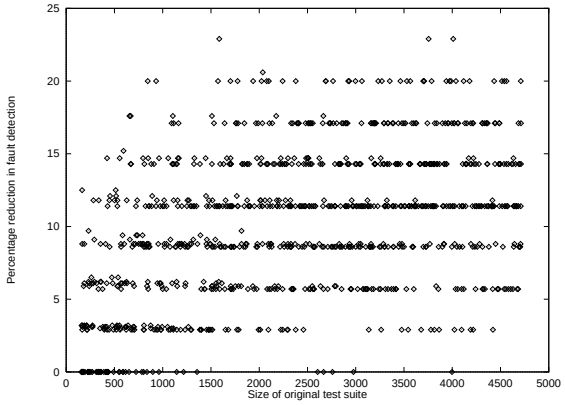

Figure 13. Percentage reduction in fault detection as a result of test suite reduction.

suites based on our reduction algorithm. Although the scatter plot shows a quadratic curve, we are investigating why there appears to be two different curves in the graph.

Because studies have shown that there is fault-detection loss in a reduced test suite, we also also considered the fault-detection capability lost by each test suite when that test suite was reduced using our algorithm. To do this, we compared the ability of the reduced test suite to detect faults in 35 faulty versions of `Space`. F denotes the number of distinct faults revealed by original test suite T over the faulty versions of program P, and F_{red} denotes the number of distinct faults revealed by reduced test suite T_{red} over those versions. The *number of faults lost* is given by ($F - F_{red}$), and the *percentage reduction in fault-detection effectiveness* of test-suite reduction is given by ($\frac{F-F_{red}}{F} * 100$). Figure 13 shows the percentage reduction in fault detection of the reduced test suite compared to the original suite. The average percentage reduction in fault detection for all of the reduced test suites is 10.2%.

6 Conclusions and Future Work

This paper has presented two new algorithms that can account for MC/DC when reducing and prioritizing test suites. The paper also presents an empirical study that evaluates some aspects of the test-suite reduction algorithm. The results achieved thus far are encouraging in that they show the potential for substantial test-suite size reduction with respect to MC/DC. Such techniques can significantly reduce the cost of regression testing for those users of this powerful testing criterion.

To evaluate our new test-suite prioritization algorithm, we are currently performing similar studies with `Space`. Our preliminary results indicate that the test-suite prioritization efficiently orders a test suite for MC/DC. We will evaluate the fault-detection capabilities of the the test-suite orderings using metrics that have been employed in previous studies [5, 11]. We are also planning experiments that will evaluate the tradeoffs between our test-suite reduction algorithm and the reduced test suite produced by our test-suite prioritization algorithm (as described in Section 4).

Although our initial studies are encouraging, much more experimentation must be conducted to verify the effectiveness of our techniques. Specifically, we are planning experiments that use our test-suite reduction and prioritization prototypes on commercial airborne software where MC/DC is required. These studies will let us evaluate our algorithms and help us provide guidelines for test-suite reduction and prioritization in practice.

Acknowledgments

This work was supported in part by a grant from Boeing Aerospace Corporation to Georgia Tech, by National Science Foundation awards CCR-9707792, CCR-9988294, CCR-0096321, and EIA-0196145 to Georgia Tech, and by the State of Georgia to Georgia Tech under the Yamacraw Mission. The anonymous reviewers provided comments that helped improve the paper's presentation.

References

[1] T. Y. Chen and M. F. Lau. Dividing strategies for the optimization of a test suite. *Information Processing Letters*, 60(3):135–141, Mar. 1996.

[2] Y. Chen, D. Rosenblum, and K. Vo. TestTube: A system for selective regression testing. In *Proc. of the 16th Int'l Conf. on Softw. Eng.*, pages 211–222, May 1994.

[3] J. J. Chilenski and S. P. Miller. Applicability of modified condition/decision coverage to software testing. *Softw. Eng. Journal*, 9(5):193–200, 1994.

[4] S. Elbaum, A. Malishevshy, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proc. of the Int'l Conference on Softw. Eng.*, pages 329–338, May. 2001.

[5] S. Elbaum, A. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. In *Proc. of the ACM Int'l Symp. on Softw. Testing and Analysis*, pages 102–112, Aug. 2000.

[6] M. R. Garey and D. S. Johnson. *Comp. and Intractability*. W.H. Freeman, New York, 1979.

[7] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Trans. on Softw. Eng. and Meth.*, 2(3):270–285, July 1993.

[8] J. Offutt, J. Pan, and J. M. Voas. Procedures for reducing the size of coverage-based test sets. In *Proc. of the 12th Int'l Conf. on Testing Comp. Softw.*, pages 111–123, June 1995.

[9] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Trans. on Softw. Eng. and Meth.*, 6(2):173–210, Apr. 1997.

[10] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Proc. of the Int'l Conf. on Softw. Maint.*, Nov. 1998.

[11] G. Rothermel, R. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Trans. on Softw. Eng. (to appear)*.

[12] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set minimization on fault detection effectiveness. *Softw. – Practice and Experience*, 28(4):347–369, Apr. 1998.

[13] W. E. Wong, J. R. Horgan, A. P. Mathur, and A. Pasquini. Test set size minimization and fault detection effectiveness: A case study in a space application. In *Proc. of the 21st Annual Int'l Comp. Softw. & Appl. Conf.*, pages 522–528, Aug. 1997.